

Assembly/Machine Language

CPE380, Fall 2026

Hank Dietz

<http://aggregate.org/hankd/>

Compiling a C Program

1. **Compiler** generates **assembly code**
 2. **Assembler** creates **binary modules**
 - Machine code, data, & symbolic info
 - Libraries are modules too
 3. **Linker** combines needed modules into one
 4. **Loader** is the part of the OS that loads a module into memory for execution
- **Usually, HLL programmers don't see this;**
1-3 invoked by **cc**, 4 when you run the program

Assembly Language(s)?

- Not one language, but *one per ISA*
- “Human readable” textual representation
 - Typically, one line becomes one instruction
 - May also have **macros**
 - **Directives** control assembly, specify data
- Used to be used for programming... now:
 - Used mostly as compiler target
 - People use it for debugging, performance tweaking, or when no other option exists

Which Assembly Language?

- Which assembly language will we use?
 - MIPS? RISC V?
 - IA32 or AMD64/Intel64/X86-64?
 - ARM?
- We'll start with a simple stack instruction set:
 - Close to what most compilers do internally
 - Can transform to whichever
- No, the stack instruction set isn't in the text...

Worlds Inside Programs

- Most programming languages are very similar, **procedural** (as opposed to **descriptive**, etc.)
- Code:
 - Assignments & expressions
 - Control flow
 - Functions & subroutines
- Data
- Comments – which we'll ignore :-)

Worlds Inside Programs

- Most programming languages are very similar, **procedural** (as opposed to **descriptive**, etc.)
- Code:
 - Assignments & expressions – **varies widely**
 - Control flow – **easy, similar in most ISAs**
 - Functions & subroutines – **complex!**
- Data – **easy, similar in most ISAs**
- Comments – which we'll ignore :-)

Control Flow

- Determines **sequence/order** of operations (orders can be **parallel**)
- HLLs have many constructs:
 - **if-then-else, switch-case**, etc.
 - **while-do, repeat-until, for**, etc.
 - **goto, break, continue**
- Most assembly languages just have goto and conditional goto... so that's what we must use to implement everything

Compilation / Translation

- Compiler “understands” program and translates it into a language the machine can execute...?
- Compilation is really based on “compiling” a bunch of code chunks that represent each part of your program into the translated constructs
- Compiler optimization isn't really “optimal” – apply correctness-preserving transformations
- Parallelizing is reordering operations; optimizing by making various things happen in parallel

Translation Templates

- It's about pattern matching & substitution
 - Patterns contain **terminals**
 - Also contain nested patterns (**nonterminals**)
- General form:

nonterminal: *{list of terminals & nonterminals}*

 {output pattern}

if (*expr*) *stat*

- *expr* and *stat* are names of other patterns
- Jump over *stat* if *expr* is false, create label

{code for expr}

Test

JumpF **L**

{code for stat}

L:

if (*expr*) *stat1* **else** *stat2*

- *stat1* and *stat2* are just *stat*
- Jump over *stat2* if *stat1* was executed

{code for expr}

Test

JumpF **L**

{code for stat1}

Jump **M**

L: *{code for stat2}*

M:

if (expr) stat1 else stat2

- There are two jumps for the then clause...
why not reorder to make that the fast case?

```
{code for expr}  
Test  
JumpT L  
{code for stat2}  
Jump M
```

```
L: {code for stat1}
```

```
M:
```

while (*expr*) *stat*

- Loop body executes 0 or more times

L: *{code for expr}*
 Test
 JumpF **M**
 {code for stat}
 Jump **L**

M:

do *stat* **while** (*expr*) ;

- Loop body executes 1 or more times
- Code is more efficient than for while loop

```
L:   {code for stat}  
      {code for expr}  
      Test  
      JumpT L
```

while (*expr*) *stat*

- Improve while by using do-like sequence enclosed in an if

```
    {code for expr}  
    Test  
    JumpF M  
L:   {code for stat}  
    {code for expr}  
    Test  
    JumpT L  
M:
```

while (*expr*) *stat*

- Improve while by jumping into loop...
nothing wrong with unstructured code here

Jump **M**

L: {*code for stat*}

M: {*code for expr*}

Test

JumpT **L**

for (*expr1*; *expr2*; *expr3*) *stat*

- Really “syntactic sugar” for:

```
expr1;  
while (expr2) {  
    stat;  
L:    expr3;  
}
```

- Only difference is **continue** goes to **L**

DO *label var=expr1, expr2, expr3*

- Fortran DO loops imply lots of stuff, e.g.:
 - Is loop counting up or down?
 - If *var* is a **real**, Fortran requires converting the index into an integer to avoid roundoff
- Implying more information is just more syntactic sugar – use a simpler source language pattern to encode a more complex, but common, target code sequence

`switch (expr) stat`

- Not equivalent to a sequence of if statements; this is C's version of a “computed goto”
- The `case` labels inside `stat` are merely labels, and so is `default`, which is why there's `break`
- Depending on case values, compilers code as:
 - Linear sequence of `if-gotos`
 - Binary search of `if-gotos`
 - Index a table of `goto` targets
 - Combinations of the above...

Assignments & Expressions

- This is where the computation happens
- Assignment notation was a major advance;
Cobol's **add c to b giving a** is **a=b+c**
- Expressions (*expr*) compute a value
- Assignments associate a value with a name:

name=expr

name=expr ?

- Not really math; it binds a value to a name
- Names (**lval**) are **places that can hold values**; registers or main memory addresses
- Expressions (**rval**, value) are **computed results**
- Consider some examples:
 - a=5** associates value 5 with name a
 - 5=a** 5 is not a name
 - a=b** associates a copy of b's value with a

a=5

- Let's generate simple stack code for this...

```
Push  a      ;push &a on stack  
Push  5      ;push the value 5  
Store        ;*(&a)=5, remove &a
```

- but where's the **;** at the end?
 - C has an *assignment operator*
 - **;** simply means discard the value produced

a=5;

Push	a	;push &a on stack
Push	5	;push the value 5
Store		;*(&a)=5, remove &a
Pop		;discard remaining 5

b = (a = 5) ;

- **b** gets a copy of **a**'s value

Push	b	; push &b on stack
Push	a	; push &a on stack
Push	5	; push the value 5
Store		; *(&a) = 5, remove &a
Store		; *(&b) = 5, remove &b
Pop		; discard remaining 5

b+c

- What does **b+c** mean – what's added?
It adds **rvals** to produce an **rval** result.
- What does **b.c** mean?
It adds **lvals** to produce an **lval** result:
&b + offset_of_field_c
- What does **b[c]** mean?
It adds **lval+rval** to produce an **lval** result:
&(b[0]) + (c * sizeof(b[c]))
- If you know which are lvals and rvals, it's easy...

a = (b + c) ;

Push	a	; push &a on stack
Push	b	; push &b on stack
Ind		; replace &b with *(&b)
Push	c	; push &c on stack
Ind		; replace &c with *(&c)
Add		; replace b, c with b+c
Store		; a=b+c, remove &a
Pop		; discard remaining b+c

a = (b + c) ;

Push	a	; push &a on stack
Push	b	; push &b on stack
Ind		; replace &b with *(&b)
Push	c	; push &c on stack
Ind		; replace &c with *(&c)
Add		; replace b, c with b+c
Store		; a=b+c, remove &a
Pop		; discard remaining b+c

`if (b+c) stat;`

```
Push  b    ;push &b on stack
Ind    ;replace &b with *(&b)
Push  c    ;push &c on stack
Ind    ;replace &c with *(&c)
Add    ;replace b, c with b+c
Test   ;tests and pops
JumpF L
{code for stat}
```

L:

`if (b<c) stat;`

```
Push  b    ;push &b on stack
Ind    ;replace &b with *(&b)
Push  c    ;push &c on stack
Ind    ;replace &c with *(&c)
Lt     ;replace b, c with b<c
Test   ;tests and pops
JumpF  L
{code for stat}
```

L:

a = (b + (5 * c)) ;

Push	a	;push &a on stack
Push	b	;push &b on stack
Ind		;replace &b with *(&b)
Push	5	;push 5 on stack
Push	c	;push &c on stack
Ind		;replace &c with *(&c)
Mul		;5, c becomes 5*c
Add		;b, 5*c becomes b+5*c
Store		;a=b+5*c, remove &a
Pop		;discard b+5*c

a=b[c];

Push	a	;push &a on stack
Push	b	;push &b on stack
Push	c	;push &c on stack
Ind		;replace &c with *(&c)
Push	4	;push sizeof(b[c])
Mul		;c, 4 becomes c*4
Add		;&b, c*4 becomes &b+c*4
Ind		;&(b[c]) becomes b[c]
Store		;a=b[c], remove &a
Pop		;discard b[c]

Different Models

- **Stack** code – easy to generate, as you saw...
- **General Register** code
 - 3 operand (MIPS): $reg1 = reg2 \text{ op } reg3$
 - 2 operand (IA32): $reg1 = reg1 \text{ op } reg3$
 - accumulator: $acc = acc \text{ op } mem$
- **Load/Store** vs. memory operands:
 $reg1 = reg1 \text{ op } mem$
- HLL-oriented **Memory-to-Memory** (IAPX432):
e.g., $a[i] = b[j] * c[k]$

a=b[c];

Push	a	;stack: &a
Push	b	;stack: &a, &b
Push	c	;stack: &a, &b, &c
Ind		;stack: &a, &b, c
Push	4	;stack: &a, &b, c, 4
Mul		;stack: &a, &b, c*4
Add		;stack: &a, &(b[c])
Ind		;stack: &a, b[c]
Store		;stack: b[c]
Pop		;stack:

a=b[c];

Push	a	; r0=&a
Push	b	; r0=&a, r1=&b
Push	c	; r0=&a, r1=&b, r2=&c
Ind		; r0=&a, r1=&b, r2=c
Push	4	; r0=&a, r1=&b, r2=c, r3=4
Mul		; r0=&a, r1=&b, r2=c*4
Add		; r0=&a, r1=&(b[c])
Ind		; r0=&a, r1=b[c]
Store		; r0=b[c]
Pop		

a=b[c];

Push	a	; r0=&a	Li	r0, a
Push	b	; r1=&b	Li	r1, b
Push	c	; r2=&c	Li	r2, c
Ind		; r2=c	Lw	r2, @r2
Push	4	; r3=4	Li	r3, 4
Mul		; r2=c*4	Mul	r2, r2, r3
Add		; r1=&(b[c])	Add	r1, r1, r2
Ind		; r1=b[c]	Lw	r1, @r1
Store		; r0=b[c]	Sw	r1, @r0
Pop				

Two Vs. Three Operands

- Uses fewer instruction bits...
MIPS three of 32 registers takes $3 \times 5 = 15$ bits;
IA32 two of 8 registers takes $2 \times 3 = 6$ bits
- From stack code, it doesn't cost anything
- With a smart compiler avoiding recomputation (e.g., via **common subexpression elimination**), might need to fake three operands:

Op r1, r2, r3 *becomes* Mov r1, r2
Op r1, r3

Two Vs. Three Operands

```
Li r0, a
Li r1, b
Li r2, c
Lw r2, @r2
Li r3, 4
Mul r2, r2, r3
Add r1, r1, r2
Lw r1, @r1
Sw r1, @r0
```

```
Li r0, a
Li r1, b
Li r2, c
Lw r2, @r2
Li r3, 4
Mul r2, r3
Add r1, r2
Lw r1, @r1
Sw r1, @r0
```

Load/Store Vs. Mem Operands

- Easier to build pipelined implementation if load/store are the only memory accesses (as in RISC architectures like MIPS)
- Memory used to be faster and processor couldn't fit lots of registers...
 - Memory operands mean fewer instructions
 - Pairs well with two operand forms (IA32)
 - Accumulator must allow memory operands (where else to get second operand?)

Load/Store Vs. Mem Operands

Load/Store

2 Operand
with Mem

Accumulator
with Mem

Li r0, a

Li r1, b

Lw r1, @r1

Li r2, c

Lw r2, @r2

Add r1, r1, r2

Sw r1, @r0

Lw r0, @b

Add r0, @c

Sw r0, @a

Lw @b

Add @c

Sw @a

How Many Registers Needed?

Li r0, a	; 1 register
Li r1, b	; 2 registers
Li r2, c	; 3 registers
Lw r2, @r2	; 3 registers
Li r3, 4	; 4 registers
Mul r2, r2, r3	; 4 registers
Add r1, r1, r2	; 3 registers
Lw r1, @r1	; 2 registers
Sw r1, @r0	; 2 registers

Spill/Reload Fakes More

```
Li r0,a
Li r1,b
Li r2,c
Lw r2,@r2
Li r3,4
```

```
Mul r2,r2,r3
Add r1,r1,r2
Lw r1,@r1
Sw r1,@r0
```

```
Li r0,a
Li r1,b
Li r2,c
Lw r2,@r2
{ Spill t0=r0 }
Li r0,4
Mul r2,r2,r0
Add r1,r1,r2
Lw r1,@r1
{ Reload r0=t0 }
Sw r1,@r0
```

HLL Memory-to-Memory

- Advantages:
 - Easier to write complex assembly code
(but we use compilers for that now and this actually makes the compiler harder to write)
 - Can enforce strict typing, software reliability
(but complicates hardware a lot)
 - Allows glueless parallel processing by keeping all program state in memory
(but memory access is s-l-o-w)
- IAPX432 did this... nothing since then

Parallel Machines

- There are two flavors of large-scale parallelism:
 - **MIMD**: different program on each PE
(multi-core processors, clusters, etc.)
 - **SIMD**: same instruction on PE's local data
(**GPUs** – graphics processing units)
- Each MIMD PE runs a sequential program...
nothing special in code generation
- **SIMD machines are different**:
 - If one PE executes some code, all must
 - Can **disable** a PE that doesn't want to do it

SIMD Code

- There are two flavors of data
 - **Singular, Scalar**: one value all PEs agree on
 - **Plural, Parallel**: value local to each PE
- Assignments and expressions work normally, except when mixing singular and plural:
 - Singular values can be copied to plurals
 - Plural values have to be “reduced” to a single value to treat as singular; for example, using operators like **any** or **all**
- Control flow is complicated by **enable masking...**

if (*expr*) *stat*

- Jump over *stat* if *expr* is false for all PEs; otherwise, do for all the PEs where it's true

PushEn	;save PE enable state
<i>{code for expr}</i>	
Test	;test on each PE...
DisableF	;turn off if false
Any	;any PE still enabled?
JumpF L	;any PE must do stat?
<i>{code for stat}</i>	
L: PopEn	;restore enable state

```
if (c < 5) a = b;
```

- Masking idea can be used in sequential code to avoid using control flow: **if conversion**
- The above can be rewritten as:

```
a = ((c < 5) ? b : a);
```

- Bitwise AND with -1 can be used to enable, while AND with 0 disables, thus simply OR:

```
t = -(c < 5);  
a = ((t & b) | ((~t) & a));
```

while (*expr*) *stat*

- Keep doing *stat* while *expr* is true for any PE; once off, PE stays off until while ends

```

    PushEn                ;save PE enable state
M:   {code for expr}
    Test                  ;test on each PE...
    DisableF              ;turn myself off if false
    Any                   ;any PE still enabled?
    JumpF L               ;exit if no PE enabled
    {code for stat}
    Jump M
L:   PopEn                ;restore enable state
```

Functions & Subroutines

- Mixes expressions and control flow...
- Complex!
 - Support of recursion
 - Lots of stuff that has to happen
 - Each ISA does it a little differently... but specifies it (e.g., as part of the ABI)
- We'll focus on generically what must happen

Simple Subroutine Call/Return

- Jump, but first save **return address** on stack

```
sub () ;
```

```
Push L  
Jump sub
```

```
L: ...
```

```
sub () {  
    ...  
    return;  
}
```

```
sub: ...  
Ret ; PC=pop
```

Simple Subroutine Call/Return

- Jump, but first save return address on stack
- Very common, and **L** is actually PC value when executing, so often a special instruction:

```

          Push L          Call sub
          Jump sub
L:        ...
```

Stack Frame

- The return address isn't all we must pass...
- Everything for a particular call is a **stack frame**:
 - Return address
 - Return value (for a function)
 - Argument values
 - Local variables
 - Temporaries
 - *Optionally*, a **frame pointer (FP)**
- Call/return and stack use is specified in ABI

Function Call

- Reserve space for **return value** first...
- Then push args & remove them on return

```
a = f(5);
```

```
Push a
Push 0 ;ret value
Push 5 ;push arg
Call f
Pop    ;pop arg
Store
Pop
```

Function Call

```
f(int b) {  
    return(b+1);  
}
```

```
f:  Push 16  
    ASP  
    Push 16  
    ASP  
    Ind  
    Push 1  
    Add  
    Store  
    Pop  
    Ret
```

Function Call

```
f:  Push 16 ;offset of ret value (0)
     ASP      ;add stack pointer
     Push 16 ;stack offset of b
     ASP
     Ind       ;get rval of b
     Push 1 ;add 1
     Add
     Store    ;store into ret value
     Pop      ;remove extra copy
     Ret
```

Frame Pointer

- Where did the stack offsets come from?
 - Subsequent pushing onto stack changes offset!
- f:** **Push 16** ; stack offset of ret value
 ...
 Push 16 ; stack offset of **b**
- **Frame pointer (FP)** points at a fixed point in the stack (saved **FP**), forming a linked list of frames

Function Call Using FP

- **Mark** pushes old FP, makes new FP point at it
- **Release** restores old FP, removes frame

```
a = f(5);
```

```
Push a
Push 0 ;ret value
Push 5 ;push arg
Mark
Call f
Release
Pop    ;pop arg
Store
Pop
```

Function Call Using FP

```
f(int b) {  
    return(b+1);  
}
```

```
f:  Push 4 ;always f  
    AFP  
    Push -4 ;always b  
    AFP  
    Ind  
    Push 1  
    Add  
    Store  
    Pop  
    Ret
```

What Is Passed For Args?

- **Call by value**: copy of rval
 - used by most languages (C, Java, etc.)
 - **considered safest way to pass values**
- **Call by address or reference**: copy of lval
 - used by: ForTran, C* reference, Pascal var
 - **efficiently avoids copying big data structures**
- **Call by name or thunk**: pointer to function to compute lval as it would have thunk to earlier
 - used by: Algol, some Lisp variants
 - **interesting, but strange and dangerous**

The Operating System (OS)?

- Trusted code that is always present to control resource allocation at runtime; it is *privileged* to touch all hardware
- Invoked by a privileging “call” to trusted code
 - User program issues a **system call**
 - **Interrupt** from an I/O device (e.g., timer)
- OS “return” removes privilege, can return to a place it didn’t come from (e.g., **timesharing**)

Enough Generalization: MIPS!

- We'll be using MIPS throughout this course (and RISC V is a lot like MIPS!)
- A simple, 32-bit, RISC architecture:
 - 32 general registers, 3-register operands
 - Strict load/store for memory access
 - Every instruction is one 32-bit word
 - Memory is byte addressed (4 bytes/word)
 - Closely matched to the C language

MIPS Registers (\$ names)

\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0-\$v1	2-3	value results
\$a0-\$a3	4-7	arguments (not on stack)
\$t0-\$t9	8-15, 24-25	temporaries
\$s0-\$s7	16-23	save before use
\$k0-\$k1	26-27	reserved for OS kernel
\$gp	28	global pointer (const)
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

MIPS ALU Instructions

- Either 3 reg operands or 2 regs and immediate 16-bit value (sign extended to 32 bits):

add	\$rd, \$rs, \$rt	$\#rd = rs + rt$
addi	\$rt, \$rs, immed	$\#rt = rs + immed$

- Suffix of **i** means immediate (**u** for unsigned)
- The usual operations: **add**, **sub**, **and**, **or**, **xor**
- Also has set-less-than, **slt**: $rd = (rs < rt)$

MIPS Load Immediate

- Can directly load a 16-bit immediate:

```
addi $rt,$0,imm    #rt=0+imm
```

- For 32-bit, generally use 2 instructions to load upper 16 bits then OR-in lower 16 bits:

```
lui $rt,imm        #rt=(imm<<16)
```

```
ori $rt,$rs,imm    #rt=rs|(imm&0xffff)
```

- MIPS assembler macro does it as **li** or **la**:

```
li $dest,const     #dest=const
```

MIPS Load & Store

- Can access a memory location given by a register plus a 16-bit Immediate offset:

lw	\$rt, off(\$rs)	#rt=memory[rs+off]
sw	\$rt, off(\$rs)	#memory[rs+off]=rt

- Byte and halfword using **b** and **h** instead of **w**

MIPS Jumps

- MIPS has a jump instruction, **j**:

j address #PC=address

- Call saves return address in **\$ra**: **jal addr**
- Return is jump register using **jr \$ra**
- Limited range (26 bits) for **j** or **jal**;
can do full 32-bit target using jump register:

la \$t0, address #t0=address
jr \$t0 #PC=t0

MIPS Branches

- MIPS has only conditional branches:

beq **\$rs**, **\$rt**, **lab** #if $rs == rt$, $PC = lab$
bne **\$rs**, **\$rt**, **lab** #if $rs \neq rt$, $PC = lab$

- The target is encoded as a 16-bit immediate:

immediate = **(lab - (PC+4)) >> 2**

- Branch over jump to target distant address

MIPS Comparisons

- Truth in C is “non-0,” so compare to **\$0**
- Equality comparison can use **xor** or **sub**
- Inequality comparisons all use **slt**:

$\$t0 = \$t1 < \$t2$ **slt** $\$t0, \$t1, \$t2$

$\$t0 = \$t1 >= \$t2$ **!** $\$t0 = \$t1 < \$t2$

$\$t0 = \$t1 > \$t2$ **slt** $\$t0, \$t2, \$t1$

$\$t0 = \$t1 <= \$t2$ **!** $\$t0 = \$t1 > \$t2$

MIPS Assembler Notation

- One assembly directive or instruction per line
- `#` means to end of line is a comment
- Labels look like they do in C, followed by a `:`
- Directives generally start with a `.`

<code>.data</code>	<code>#the following is static data</code>
<code>.text</code>	<code>#the following is code</code>
<code>.globl name</code>	<code>#name is what C calls extern</code>
<code>.word value</code>	<code>#initialize a word to value</code>
<code>.ascii "abc"</code>	<code>#initialize bytes to 97,98,99</code>
<code>.asciiz "abc"</code>	<code>#initialize bytes to 97,98,99,0</code>

MIPS References & Tools

- Reference materials:
 - The course website
 - The textbook
 - MIPS **cc** **-S**
- Simulator we prefer is **SPIM**, WWW version:
<https://aggregate.org/cgi-bin/cgispim.cgi>
- There's even a little C-subset compiler:
<https://aggregate.org/cgi-bin/mucky.cgi>

RISC V vs. MIPS

- RISC V started at Berkeley in 2010, and is a “free and open” extendable MIPS-like ISA
<https://riscv.org/>
- Similar instructions and assembly syntax
 - Expandable instruction format, 16-bit parcels
 - Branches do comparisons, not just equality
 - `lui` uses 20 bits because immmed is 12 bits
 - Various ratified extensions: multiplication, atomics, floats, 16-bit format, vectors, etc.

RISC V

- Encoding is more complex than MIPS...

32-bit RISC-V instruction formats

Format	Bit																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]				opcode							
Branch	[12]	imm[10:5]					imm[4:1]														[11]											
Register/register	funct7																				rd											
Immediate	imm[11:0]																															
Upper immediate	imm[31:12]																															
Jump	[20]	imm[10:1]										[11]	imm[19:12]																			

- **opcode (7 bits):** Partially specifies one of the 6 types of *instruction formats*.
- **funct7 (7 bits) and funct3 (3 bits):** These two fields extend the *opcode* field to specify the operation to be performed.
- **rs1 (5 bits) and rs2 (5 bits):** Specify, by index, the first and second operand registers respectively (i.e., source registers).
- **rd (5 bits):** Specifies, by index, the destination register to which the computation result will be directed.

Summary

- There are many different assembly languages, but there are many similarities
- ISA specifies instructions (ABI for conventions)
- MIPS is a very straightforward RISC made for C
- You don't need to write lots of assembly code
 - tweak code output by a compiler
 - write little wrappers for what compiler can't do